

DOM (Document Object Model)

A Detailed Introduction

(Detailed notes yet beginner friendly)

Start with a Question:

"You have HTML and JavaScript. How do they talk to each other?"

The Setup:

```
<!-- index.html -->
<!DOCTYPE html>
<html>
<head>
  <title>My Page</title>
</head>
<body>
  <h1>Hello World</h1>
  <p>This is a paragraph.</p>
</body>
</html>
```

```
// script.js
// How do I change "Hello World" to "Goodbye World"?
// How do I make the paragraph red?
// How do I add a new button?
```

The Core Problem:

- HTML is just text with tags (markup language)
- JavaScript is a programming language (works with objects, functions, variables)
- They speak different "languages"!

Real-world Analogy:

- HTML = Blueprint of a house (static document)
- JavaScript = Construction crew (wants to modify the house)
- DOM = The actual built house that the crew can walk through and modify

Key Point: "The DOM is the bridge that lets JavaScript understand and manipulate HTML."

What is the DOM?

Definition: “The DOM (Document Object Model) is a tree-like representation of your HTML document that JavaScript can understand and manipulate”.

Visual Explanation:

Your HTML:

```
<html>
  <head>
    <title>My Site</title>
  </head>
  <body>
    <h1>Welcome</h1>
    <p>Hello there!</p>
  </body>
</html>
```

Browser Creates This Tree:

```
document
├── html
│   ├── head
│   │   └── title
│   │       └── "My Site"
│   └── body
│       ├── h1
│       │   └── "Welcome"
│       └── p
│           └── "Hello there!"
```

Key Concepts (Explain with Diagrams):

1. Everything is a Node

Types of Nodes:

- Element Nodes: <div>, <p>, <h1>
- Text Nodes: The actual text content
- Document Node: The root (document)

2. HTML Elements Become Objects

```
<h1 id="title">Hello</h1>
```

Becomes:

```
{
  tagName: "H1",
  id: "title",
  textContent: "Hello",
  style: { color: "", fontSize: "" },
  parentElement: body,
  children: [],
  // ... many more properties and methods
}
```

NOTE: "Every HTML tag becomes a JavaScript object with properties and methods!"

Draw this hierarchy:

```
Window (global object - the browser API)
├── document (your HTML page as objects)
│   ├── documentElement (the <html> tag)
│   │   ├── head
│   │   └── body
│   │       └── (all your elements)
```

Key Points:

- **window** = The browser environment (has alert, setTimeout, localStorage, etc.)
- **document** = Your HTML page (the DOM tree)

Sample HTML to Work With

```
<!DOCTYPE html>
<html>
<head>
  <title>DOM Selection</title>
</head>
<body>

  <div id="main-container" class="container">
    <h1 class="title">Main Title</h1>
    <p>This is the first paragraph.</p>
    <p class="content">This is the second paragraph with a class.
  </p>

    <ul id="item-list">
      <li class="item">Item 1</li>
      <li class="item special">Item 2</li>
      <li class="item">Item 3</li>
    </ul>

    <form name="login-form">
      <input type="text" name="username">
    </form>
  </div>

  <div class="container footer">
    <p>Footer content.</p>
  </div>

</body>
</html>
```

1. The Classic, Specific Methods (The Old Guard)

These were the original ways to select elements. They are very fast for their specific purpose but are less flexible than the modern methods.

A. `document.getElementById('id')`

- **What it does:** Selects the single element that has the specified **id**.
- **Returns:** A single element object, or **null** if no element with that ID is found.
- **First Thought:** "Get me the one, unique thing with this exact ID."

```
const mainContainer = document.getElementById('main-container');  
mainContainer.style.border = '2px solid red'; // Puts a red border  
around the main div
```

```
const itemList = document.getElementById('item-list');  
console.log(itemList);
```

Why it's great: It's extremely fast and direct because IDs are meant to be unique in a document. This is the best method to use when you have a unique ID.

B. `document.getElementsByTagName('tagName')`

- **What it does:** Selects all elements that have the specified tag name (like **p**, **li**, **div**).
- **Returns:** **HTMLCollection** (an array-like object) of all matching elements.
- **First Thought:** "Get me all the paragraphs" or "Get me all the list items."

```
const allParagraphs = document.getElementsByTagName('p');
console.log(allParagraphs.length); // 3

// You can loop through the collection
for (let i = 0; i < allParagraphs.length; i++) {
  allParagraphs[i].style.fontStyle = 'italic';
}
```

What live means: If you add a new `<p>` to the page after you've selected them, the `allParagraphs` collection will automatically update to include it.

C. `document.getElementsByClassName('className')`

- **What it does:** Selects all elements that have the specified class name.
- **Returns:** A live **HTMLCollection** of all matching elements.
- **First Thought:** "Give me everything with the class 'item'."

```
const allItems = document.getElementsByClassName('item');  
console.log(allItems.length); // 3
```

```
// You can also get elements with multiple classes  
const footerContainer =  
document.getElementsByClassName('container footer');  
console.log(footerContainer[0]);
```

2. The Modern, Powerful Methods (The "Query" Selectors)

These methods were a game-changer. They allow you to select elements using the same powerful CSS selector syntax that you use in your stylesheets. These are the methods you should use most of the time.

A. `document.querySelector('cssSelector')`

- **What it does:** Selects the **first element** in the document that matches the specified CSS selector.
- **Returns:** A **single element object**, or **null** if no match is found.
- **First Thought:** "Find me the *very first* thing that matches this CSS rule."

```
// Get the element with the ID 'main-container'
const main = document.querySelector('#main-container');

// Get the FIRST element with the class 'container'
const firstContainer = document.querySelector('.container');
console.log(firstContainer); // This will be the main div, not the footer

// Get the FIRST paragraph
const firstP = document.querySelector('p');
console.log(firstP);

// Get the list item that has BOTH 'item' and 'special' classes
const specialItem = document.querySelector('.item.special');
specialItem.style.color = 'orange';

// Get the input with the name 'username'
const usernameInput =
document.querySelector('input[name="username"]');
```

B. `document.querySelectorAll('cssSelector')`

- **What it does:** Selects all elements in the document that match the specified CSS selector.
- **Returns:** A **static NodeList** (an array-like object) of all matching elements.
- **First Thought:** "Find me *every single thing* that matches this CSS rule."

```
// Get ALL elements with the class 'container'
const allContainers = document.querySelectorAll('.container');
console.log(allContainers.length); // 2

// Get ALL list items inside the element with the ID 'item-list'
const listItems = document.querySelectorAll('#item-list li');

// A NodeList has a .forEach method, which is very convenient!
listItems.forEach(item => {
  item.style.fontWeight = 'bold';
});
```

What "static" means: Unlike an **HTMLCollection**, a **NodeList** returned by **querySelectorAll** is **not live**. If you add a new matching element to the page later, the **list items** collection will not automatically update. This is usually the behavior you want, as it's more predictable.

3. Other, More Niche Selections

- **document.getElementsByName('name')** : Selects elements based on their name attribute (most commonly used for form elements).

```
const loginForm = document.getElementsByName('login-form');
const usernameInputByName =
document.getElementsByName('username');
```

- **Traversing the DOM:** Once you have one element, you can navigate to its relatives.

```
const itemList = document.getElementById('item-list');

const parentContainer = itemList.parentElement; // The #main-
container div
const listChildren = itemList.children; // An HTMLCollection of the 3 <li>
elements
const firstLi = itemList.firstElementChild; // The first <li>
const lastLi = itemList.lastElementChild; // The last <li>

const specialLi = document.querySelector('.special');
const nextItem = specialLi.nextElementSibling; // The 3rd <li>
const prevItem = specialLi.previousElementSibling; // The 1st <li>
```

Summary: Which Method Should I Use?

Goal	Best Method	Why?
Get a single, unique element	<code>getElementById()</code>	Fastest and most direct.
Get the first element that matches a complex rule	<code>querySelector()</code>	Extremely flexible, uses familiar CSS syntax. Your default choice for single elements.
Get all elements that match a complex rule	<code>querySelectorAll()</code>	Extremely flexible, and the returned <code>NodeList</code> has a convenient <code>.forEach()</code> method. Your default choice for multiple elements.
Get all elements by tag or class (and you need a "live" collection)	<code>getElementsByName()</code> or <code>getElementsByClassName()</code>	Use these only if you specifically need the "live" auto-updating behavior. Otherwise, <code>querySelectorAll</code> is better.

Once you've selected an element, the next step is to interact with it. You do this by reading and writing to its properties. These properties are the bridge that lets your JavaScript code control the content, appearance, and attributes of the HTML elements.

Let's do a deep dive into the most important properties, using this HTML snippet as our reference:

```
<div id="product-card" class="card featured">
  <h2 id="product-name">
    Smart Headphones
    <!-- This is a comment -->
    <span style="display: none;">SALE</span>
  </h2>
  <p id="description">
    These headphones have <strong>noise-cancelling</strong> features.
  </p>
</div>
```

1. Properties for Manipulating Content

These three properties are used to read or change the content inside an element, but they have critical differences.

A. `.textContent` (The Safe and Recommended Choice)

- **What it does:** It gets or sets the raw text content of an element and all its descendants. It completely ignores all HTML tags and gives you just the text.
- **First Thought:** "Give me just the text, exactly as it is, with no HTML."
- **When Reading:**

```
const desc = document.getElementById('description');
console.log(desc.textContent);
// Output: "These headphones have noise-cancelling features."
// Notice the <strong> tags are gone.

const productName = document.getElementById('product-name');
console.log(productName.textContent);
// Output: "Smart Headphones SALE"
// It includes text from hidden elements but ignores comments.
```


- **When Writing (Safe)** : When you set **.textContent** , the browser treats your input as pure text. It will **not** parse any HTML tags. This is a crucial security feature that prevents Cross-Site Scripting (XSS) attacks.

```
const desc = document.getElementById('description');  
  
// Let's try to inject some HTML  
desc.textContent = "Click <a href='#>here</a> to win!";  
  
// The browser will display the literal text, not a clickable link:  
// "Click <a href='#>here</a> to win!"
```

- **Performance:** It's very fast because the browser doesn't need to parse HTML.
- **Best For:** Reading or writing plain text content. This should be your default choice.

B. `.innerHTML` (The Powerful but Dangerous Choice)

- **What it does:** It gets or sets the full HTML content of an element.
- **First Thought:** "Give me **everything inside**, including all the HTML markup."
- **When Reading:**

```
const desc = document.getElementById('description');
console.log(desc.innerHTML);
// Output: "These headphones have <strong>noise-
cancelling</strong> features."
// It includes the HTML tags as a string.
```

- **When writing (Dangerous):** When you set `.innerHTML`, the browser will parse your string and create actual HTML elements from it. This is powerful, but it's a major security risk if the string comes from a user.

```
const desc = document.getElementById('description');

// This is powerful and useful for creating new elements.
desc.innerHTML = "Updated features: <strong>Active Noise
Cancelling</strong> and <em>Bluetooth 5.0</em>.";
// The browser will correctly render the bold and italic text.

// SECURITY RISK: What if the string comes from a malicious user?
let userInput = ``;
// desc.innerHTML = userInput; // This would execute the malicious script!
```

C. `.innerText` (The "Smart" but Tricky Choice)

- **What it does:** This is a non-standard but widely supported property. It tries to get the text content as it is rendered to the user, taking CSS into account.
- **First Thought:** "Give me the text **as it appears on the screen.**"
- **When Reading:**

```
const productName = document.getElementById('product-name');
console.log(productName.innerText);
// Output: "SMART HEADPHONES" (if CSS `text-transform:
uppercase` was applied)
// It will NOT include the text from the hidden <span> ("SALE").
```

- `.innerText` is "style-aware." It won't return text from hidden elements, and it might reflect CSS transformations.
- **When Writing:** It behaves similarly to `.textContent`, setting the raw text.
- **Performance:** It's the slowest of the three because the browser may need to trigger a layout calculation (a "reflow") to figure out what is actually visible on the screen.
- **Best For:** Rarely needed. Use it only if you have a specific need to get the text exactly as a user would see it, excluding hidden content. Prefer `.textContent` in almost all cases.

Summary of Content Properties:

.innerText	No	Yes	Slowest	Safe
.innerText	No	Yes	Slowest	Safe
.innerText	No	Yes	Slowest	Safe
.innerText	No	Yes	Slowest	Safe

1. Properties for Manipulating Attributes

These properties give you direct access to the HTML attributes of an element.

A. **.id** and **.className** (Direct Properties)

For the most common attributes like `id` and `class`, JavaScript provides direct properties.

- **id**: Gets or sets the `id` attribute.

```
const card = document.getElementById('product-card');  
console.log(card.id); // "product-card"  
card.id = 'new-card-id'; // Changes the element's ID
```

- **.className**: Gets or sets the entire **class** attribute as a single string. Because it overwrites everything, **.className** is clumsy. The **.classList** property is much better.

```
console.log(card.className); // "card featured"  
  
// This will OVERWRITE all existing classes.  
card.className = "card-dark-mode";  
// The element now only has the class "card-dark-mode". "featured"  
is gone.
```

B. `.classList` (The Modern Way to Handle Classes)

- **First Thought:** "Give me a smart toolbox for adding, removing, and checking for classes without messing up the other ones."
- **What it is:** An object with helpful methods to manage an element's classes.
 - `.add('className')`: Adds a new class.
 - `.remove('className')`: Removes a class.
 - `.toggle('className')`: Adds the class if it's missing, removes it if it's present.
 - `.contains('className')`: Returns **true** or **false** if the element has the class.

```
const card = document.getElementById('product-card');

card.classList.add('in-cart');    // Adds 'in-cart'
card.classList.remove('featured'); // Removes 'featured'

// Toggle a 'selected' class every time a function is called
card.classList.toggle('selected');

if (card.classList.contains('in-cart')) {
  console.log("This item is in the cart.");
}
```

- **Best For:** This is the correct and modern way to manipulate CSS classes.

C. `.getAttribute()` and `.setAttribute()` (For Any Attribute)

These are generic methods that can work with any HTML attribute, including custom ones.

```
const card = document.getElementById('product-card');

// Add a custom data attribute
card.setAttribute('data-product-id', 'xyz-123');

// Get the value of an attribute
const productId = card.getAttribute('data-product-id');
console.log(productId); // "xyz-123"

// Remove an attribute
card.removeAttribute('class');
```

3. The .style Property

- **First Thought:** "Give me direct control over the element's inline styles."
- **What it does:** An object that represents the style="..." attribute of an element. You can change CSS properties through it.
- **The "Gotcha":** Property Names are camelCased. CSS properties with hyphens (like background-color) must be written in camelCase in JavaScript.
 - background-color -> backgroundColor
 - font-size -> fontSize
 - z-index -> zIndex

```
const card = document.getElementById('product-card');

// Add a custom data attribute
card.setAttribute('data-product-id', 'xyz-123');

// Get the value of an attribute
const productId = card.getAttribute('data-product-id');
console.log(productId); // "xyz-123"

// Remove an attribute
card.removeAttribute('class');
```


3. The `.style` Property

- **Important:** The `.style` property only knows about inline styles. It cannot read styles that are set in an external CSS file. To do that, you need to use the global function `window.getComputedStyle(element)`.
- **Best Practice:** It's generally better to use `.classList` to add or remove CSS classes that are defined in your stylesheet, rather than manipulating inline styles directly with JavaScript. This keeps your styling rules separate from your logic.